11장. 다양한 코틀린 클래스

11.1. 데이터 클래스

11.1.1. 데이터 클래스란?

- VO(Value-Object) 클래스
- data 라는 예약어로 선언되는 클래스

- 주생성자가 선언되어야 하며 주생성자의 매개변수는 최소 하나 이상이 선언되어 있어야 한다.
- 모든 주생성자의 매개변수는 var 혹은 val 로 선언되어야 한다.
- 데이터 클래스는 abstract, open, sealed, inner 등의 예약어를 추가할수 없다.

```
data class User(val name: String, val age: Int)
```

```
data class User1()//error

data class User2(name: String)//error

data abstract class User3(val name: String)//errordata class User4(val name: String, no: Int)//error
```

11.1. 데이터 클래스

11.1.2. 데이터 클래스의 함수

equals()
- 객체의 데이터가 같은지에 대한 비교

```
class Product(val name: String, val price:Int)

data class User(val name: String, val age: Int)

fun main(args: Array<String>) {

    var product=Product("prod1",100)
    var product1=Product("prod1",100)
    println(product.equals(product1))

    var user=User("kkang",30)
    var user1=User("kkang",30)
    println(user.equals(user1))
}
```

11.1. 데이터 클래스

```
data class User(val name: String, val age: Int)

data class Person(val name: String, val age: Int)

fun main(args: Array<String>) {
    val user=User("kkang", 20)
    val person=Person("kkang", 20)

    println(user.equals(person)) //false
}
```

- equals 함수에 의한 값의 비교는 주생성자에 선언된 프로퍼티 값만을 비교

```
data class User(val name: String, val age: Int){
    var email: String = "a@a.com"
}


fun main(args: Array<String>) {
    val user = User("kkang", 20)

    val user1 = User("kkang", 20)
    user1.email = "b@b.com"

    println(user.equals(user1))
}
```

11.1. 데이터 클래스

toString()

- 데이터 클래스의 데이터를 문자열로 반환하는 함수

```
class Product(val name: String, val price:Int)

data class User(val name: String, val age: Int){
    var email: String = "a@a.com"
}

fun main(args: Array<String>) {

    var product=Product("prod1",100)
    println(product.toString())

    var user=User("kkang",30)
    println(user.toString())
}
```

실행결과

```
fourteen_four.Product@5e2de80c
User(name=kkang, age=30)
```

11.1. 데이터 클래스

componentN()

- 클래스 프로퍼티 값을 획득

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {

    var user=User("kkang",30)

    println(user.component1()) //kkang
    println(user.component2()) //30
}
```

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {
   var user=User(age=30, name="kkang")
    val (name, age) = user

    println("name : $name, age: $age") //name : kkang, age: 30
}
```

11.1. 데이터 클래스

copy()

- 객체를 복사

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {
    var user=User(age=30, name="kkang")
    println(user.toString()) //User(name=kkang, age=30)

    var user2=user.copy(name="kim")
    println(user2.toString()) //User(name=kim, age=30)
}
```

11.2. Enum 클래스

11.2.1. 열거형 클래스 선언 및 이용

- 상수 여러 개를 열거형 타입에 선언하고 이 열거형 타입으로 선언되는 변수는 선언된 상수값중 하나를 지정하게 하는 기법.
- enum 이라는 예약어로 만들어지는 클래스

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
fun main(args: Array<String>) {
    val direction: Direction = Direction.NORTH
    println("${direction.name} ... ${direction.ordinal}")
    val directions: Array<Direction> = Direction.values()
    directions.forEach { t -> println(t.name) }
    val direction1=Direction.valueOf("WEST")
    println("${direction1.name} .. ${direction1.ordinal}")
}
```

실행결과

```
NORTH ... 0
NORTH
SOUTH
WEST
EAST
WEST .. 2
```

11.2. Enum 클래스

개발자 임의 데이터 삽입

```
enum class Direction(val no: Int) {
    NORTH(0), SOUTH(1), WEST(2), EAST(3)
}

fun main(args: Array<String>) {
    val direction: Direction = Direction.NORTH
    println(Direction.NORTH.no) //0
}
```

```
enum class Direction(var no: Int, val str: String) {
    NORTH(0, "north"), SOUTH(1, "south"), WEST(2, "west"), EAST(3, "east")
}

fun main(args: Array<String>) {
    val direction: Direction = Direction.NORTH

    println("no : ${direction.no}, ${direction.str}") //no : 0, north

    direction.no=10

    println("no : ${direction.no}, ${direction.str}") //no : 10, north
}
```

11.2. Enum 클래스

11.2.2. 익명 클래스 이용

- 열거상수는 객체
- 이름없는 클래스를 직접 정의하여 다양하게 이용
- 세미콜론 (;) 앞부분이 문자열의 나열이고 뒷 부분이 클래스에 선언하고자 하는 프로퍼티와 함수 선언 부분

```
enum class Direction {
    NORTH {
        override val data1: Int = 10
        override fun myFun(){
            println("north myFun....")
        }
    },
    SOUTH {
        override val data1: Int = 20
        override fun myFun(){
            println("south myFun....")
        }
    };
    abstract val data1: Int
    abstract fun myFun()
}
fun main(args: Array<String>) {
    val direction: Direction = Direction.NORTH
    println(direction.data1)
    direction.myFun()
}
```

11.3. Sealed 클래스

- sealed 예약어로 선언되는 클래스
- 열거형 클래스에 대한 이해를 바탕

```
sealed class Shape {
    class Circle(val radius: Double) : Shape()
    class Rect(val width: Int, val height: Int) : Shape()
}

class Triangle(val bottom: Int, val height: Int): Shape()

fun main(args: Array<String>) {
    val shape1: Shape = Shape.Circle(10.0)

    val shape2: Shape = Triangle(10, 10)

}
```

11.3. Sealed 클래스

- 열거상수에 데이터를 추가할수 있지만 모든 열거상수에 동일한 형태의 데이터만 추가가 가능
- 열거클래스는 열거상수를 Anonymous Class를 이용하지만 Sealed 클래스는 Sealed 클래스의 서브 클래스를 일반 클래스 정의하듯이 똑같이 정의해서 사용

11.4. Nested 클래스

- Nested 클래스는 특정 클래스 내에 선언된 클래스를 지칭

```
class Outer {
    class Nested {
        val name: String = "kkang"
        fun myFun(){
            println("Nested.. myFun...")
        }
    }
}

fun main(args: Array<String>) {
    val obj: Outer.Nested = Outer.Nested()
    println("${obj.name}")
    obj.myFun()
}
```

11.4. Nested 클래스

Outer 클래스의 맴버 접근

```
class Outer {
    var no: Int = 10
    fun outerFun() {
        println("outerFun()...")
    }
    class Nested {
        val name: String = "kkang"
        fun myFun(){
            println("Nested.. myFun...")
            no=20//error
            outerFun()//error
        }
    }
}
```

11.4. Nested 클래스

- Nested 클래스에서는 Outer 클래스의 맴버를 이용하려면 Nested 클래스 선언시 inner 라는 예약어를 추가해 주어야 한다.

```
class Outer {
    private var no: Int = 10
    fun outerFun() {
        println("outerFun()...")
    }
    inner class Nested {
        val name: String = "kkang"
        fun myFun(){
            println("Nested.. myFun...")
            no=20
            outerFun()
        }
    }
}

fun main(args: Array<String>) {
    val obj: Outer.Nested = Outer.Nested()//error
    println("${obj.name}")
    obj.myFun()
}
```

11.4. Nested 클래스

- inner가 추가된 클래스는 외부에서 객체 생성이 불가

```
class Outer {
    private var no: Int = 10
    fun outerFun() {
        println("outerFun()...")
    }
    inner class Nested {
        val name: String = "kkang"
        fun myFun(){
            println("Nested.. myFun...")
            no=20
            outerFun()
        }
    }

    fun createNested(): Nested {
        return Nested()
    }
}

fun main(args: Array<String>) {
    val obj1: Outer.Nested = Outer().Nested()
    val obj2: Outer.Nested = Outer().createNested()
}
```

11.5. Object 클래스

11.5.1. object를 이용한 익명 내부 클래스 정의

- object { } 형태로 클래스를 선언
- 클래스명이 없지만 선언과 동시에 객체가 생성
- object 클래스에는 생성자는 추가할수 없다.

```
val obj1=object {
    var no1: Int = 10
    fun myFun() {

    }
}

class Outer {
    val obj2 = object {
        var no2: Int = 0
        fun myFun() {

        }
    }
}
```

11.5. Object 클래스

멤버 이용

```
class Outer {

    private var no: Int = 0

    val myInner = object {
        val name: String = "kkang"
        fun innerFun(){
            println("innerFun....")
            no++
        }
    }

    fun outerFun(){
        myInner.name//error
        myInner.innerFun()//error
    }
}

fun main(args: Array<String>) {
    val obj=Outer()
    obj.myInner.name//error
    obj.myInner.innerFun()//error
}
```

11.5. Object 클래스

object 클래스 선언시 private을 추가

```
class Outer {

    private var no: Int = 0

    private val myInner = object {
        val name: String = "kkang"
        fun innerFun(){
            println("innerFun....")
            no++
        }
    }

    fun outerFun(){
        myInner.name
        myInner.innerFun()
    }
}

fun main(args: Array<String>) {
    val obj=Outer()
    obj.myInner.name//error
    obj.myInner.innerFun()//error
}
```

11.5. Object 클래스

11.5.2. 타입 명시로 object 이용

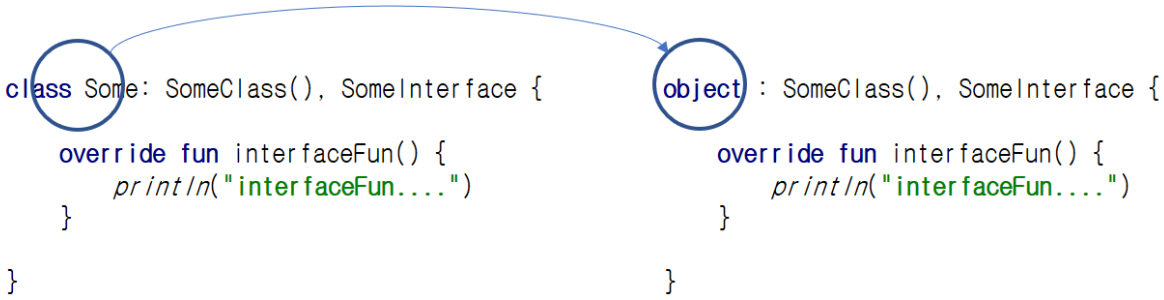
- object 클래스를 만들 때 다른 클래스를 상속 받거나 인터페이스를 구현

```
interface SomeInterface {
    fun interfaceFun()
}
open class SomeClass {
    fun someClassFun(){
        println("someClassFun....")
    }
}

class Outer {
    val myInner: SomeClass =object : SomeClass(), SomeInterface {
        override fun interfaceFun() {
            println("interfaceFun....")
        }
    }
}

fun main(args: Array<String>) {
    val obj=Outer()
    obj.myInner.someClassFun()
}
```

11.5. Object 클래스



11.5. Object 클래스

11.5.3. object 선언

- val obj = object { }
- object 클래스명 { }
- 클래스명과 동일한 이름의 객체까지 같이 생성
- object 클래스명 { } 은 객체생성구문

```
class NormalClass {
    fun myFun(){ }
}
object ObjectClass {
    fun myFun() { }
}
fun main(args: Array<String>) {
    val obj1: NormalClass = NormalClass()
    val obj2: NormalClass = NormalClass()
    obj1.myFun()

    val obj3: ObjectClass = ObjectClass()//error

    ObjectClass.myFun()
}
```

11.5. Object 클래스

11.5.4. companion 예약어

```
class Outer {
    object NestedClass {
        val no: Int = 0
        fun myFun() { }
    }
}
fun main(args: Array<String>) {
    val obj=Outer()
    obj.NestedClass.no//error

    Outer.NestedClass.no
    Outer.NestedClass.myFun()
}
```

11.5. Object 클래스

```
class Outer {
    companion object NestedClass {
        val no: Int = 0
        fun myFun() { }
    }
    fun myFun(){
        no
        myFun()
    }
}
fun main(args: Array<String>) {
    Outer.NestedClass.no
    Outer.NestedClass.myFun()

    Outer.no
    Outer.myFun()
}
```